06 Lab
Advanced mechanisms of the Scala language

Mirko Viroli, Roberto Casadei
{mirko.viroli,roby.casadei}@unibo.it

C.D.L. Magistrale in Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2020/2021

Lab 06: Outline

Outline

- Consolidate your knowledge of Scala
- Practice with advanced Scala features

Repo with exercises

- Fork/clone `https://github.com/unibo-pps/pps-20-21-lab06`
- Open the provided Scala project in IntelliJ
- The code for this lab is in package **u06lab.code**
- You may want to copy such code in a package `u06lab.solution` so that the problems and the corresponding solutions are kept separate
- For each exercise, you are given a (statically correct) code template that you have to complete as well as a main program to be executed for checking your solution and making experiments
- As usual, you may commit your changes and push them to your own (forked) repository

Exercise 1: `Combiner`

1) Implement `FunctionsImpl` such that the code in `TryFunctions` works correctly.
   - **N.B.**: complete this before looking at the following step!
2) To apply DRY principle at the best, note the three methods in Functions do something similar. Use the following approach (called type classes approach).

- find three implementations of `Combiner` that tell (for `sum`, `concat` and `max`) how to combine two elements, and what to return when the input list is empty.
  - Observe how much they are both structurally and functionally similar.
  - `Combiner[T]` is called a *type class* since it is a mechanisms to conceptually add operations to type `T`
- Implement in `FunctionsImpl` a **new method** `combine` that, other than the collection of `A`s, takes a `Combiner` object as parameter too
- Implement the three methods by simply calling `combine`
- When all works, note we completely avoided duplications.

3) Note that `combine` could take the `Combiner` implicitly

Exercise 2: `Parser`

1) Provide missing implementations such that the code in `TryParsers` works correctly.
   - Consider the `Parser` example shown in previous lesson.
   - Analogously to `NonEmpty`, create a mixin `NotTwoConsecutive`, which adds the idea that one cannot parse two consecutive elements which are equal.
   - Use it (as a mixin) to build class `NotTwoConsecutiveParser`, used in the testing code at the end.
   - Note we also test that the two mixins can work together!!
   - Write the full linearisation of `parserNTCNE`
   - **N.B.:** tests are written in such a way that each call to `parseAll` runs on a brand-new parser (got via a 0-arg `def`). If you want to avoid this (i.e., running `parseAll` multiple times on the same parser object), you need to reset the parser after use (e.g., in `parseAll`)
2) Extend Scala type `String` with a factory method that creates a parser which recognises the set of chars of a string.

Exercise 3: `TicTacToe`

Follow the exercises sketched in object `TicTacToe`.

1. Implement `find` such that the code provided behaves as suggested by the comments
2. Implement `placeAnyMark` such that the code provided behaves as suggested by the comments
3. (Advanced) Implement `computeAnyGame` such that the code provided behaves as suggested by the comments
4. (Very advanced) Modify the above one so as to stop each game when someone won